Verilog RTL for a Two-Stage SMIPSv2 Processor

6.375 Laboratory 1
February 23, 2006

For the first lab assignment, you are to write an RTL model of a two-stage pipelined SMIPSv2 processor using Verilog. The deliverables for this lab are (a) your working Verilog RTL checked into CVS and (b) written answers to the critical questions given at the end of this document. The lab assignment is due at the start of class on Friday, February 24. You are encouraged to discuss your design with others in the class, but you must turn in your own work. The two-stage pipeline should perform instruction fetch in the first stage, while the second pipeline stage should do everything else including data memory access. Since SMIPS does not have a branch delay slot, you will need to handle branches carefully to ensure that incorrect instructions are not accidental executed.

If you need to refresh your memory about pipelining and the MIPS instruction set, we recommend *Computer Organization and Design: The Hardware/Software Interface*, Second Edition, by Patterson and Hennessey. More detailed information about the SMIPS architecture can be found in the *SMIPS Processor Specification*. For more information about using Synopsys VCS for Verilog simulation consult *Tutorial 1: Simulating a SMIPSv1 Unpipelined Processor Using Synopsys VCS*.

For this assignment, you should focus on writing clean synthesizable code that follows the coding guidelines discussed in lecture. In particular, place most of your logic in leaf modules and use structural Verilog to connect the leaf modules in a hierarchy. Avoid tricky hardware optimizations at this stage, but make sure to separate out datapath and memory components from control circuitry. The system diagram in Figure 4 can be used as an initial template for your SMIPS processor implementation, but please treat it as a suggestion. Your objective in this lab is to implement the SMIPSv2 ISA, not to implement the system diagram so feel free to add new control signals, merge modules, or make any other modifications to the system.

Processor Interface

Your processor should be in a module named `smipsProc` and must have the interface shown in Figure 2. We have provided you with a test harness that will drive the inputs and check the outputs of your design. The test harness includes the data and instruction memories. We have provided separate instruction and data memory ports to simplify the construction of the two stage pipeline, but both ports access the same memory space. The memory ports can only access 32-bit words, and so the lowest two bits of the addresses are ignored (i.e., only `imemreq_bits_addr[31:2]` and `dmemreq_bits_addr[31:2]` are significant). To make an instruction memory request, set `imemreq_bits_addr` to the fetch address and set `imemreq_val` to one. The data will be returned combinationally (i.e. there are no clock edges between when a request is made and when the response returns). To make a data memory request set `dmemreq_bits_rw` to zero for a load or one for a store, set `dmemreq_bits_addr` to the address, set `dmemreq_bits_data` to the store data if needed, and finally set `dmemreq_val` to one. The data will be returned combinationally for loads, while for stores the data will be written at the end of the current clock cycle. Notice that the data write bus is a separate unidirectional bus from the data read bus. Bidirectional tri-state buses are usually avoided on chip in ASIC designs.

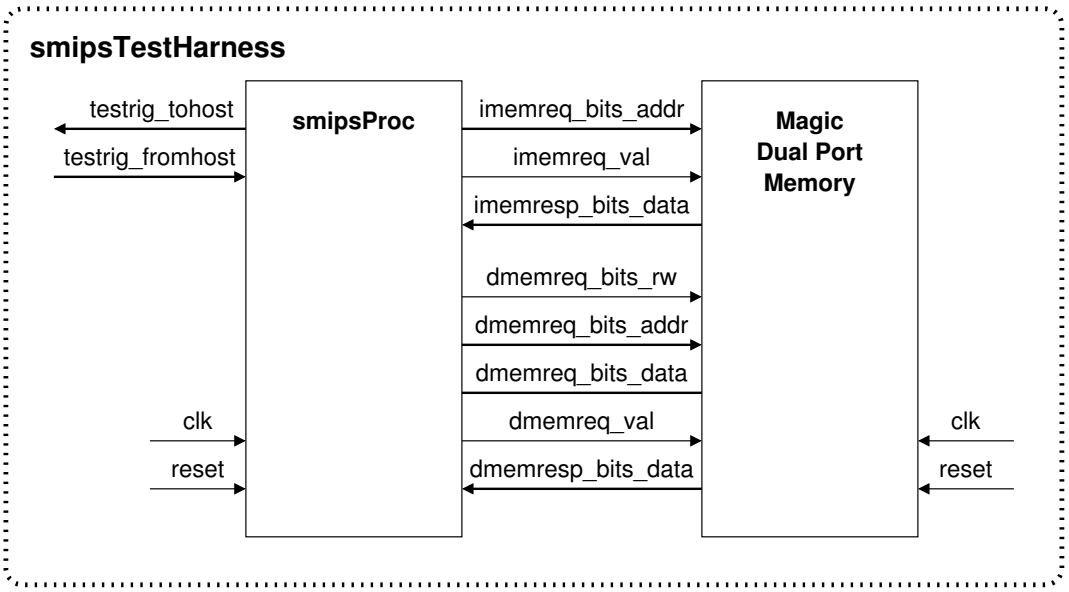

Figure 1: Block diagram for SMIPSv2 Processor Test Harness

```
module smipsProc
(

  input clk, reset,

  input  [ 7:0] testrig_fromhost,    // Testrig fromhost port
  output [ 7:0] testrig_tohost,      // Testrig tohost port (must reset to zero)

  output [31:0] imemreq_bits_addr,   // Inst mem port: addr to fetch
  output        imemreq_val,         // Inst mem port: is imem request valid?
  input  [31:0] imemresp_bits_data,  // Inst mem port: returned instruction

  output        dmemreq_bits_rw,     // Data mem port: read or write (r=0/w=1)
  output [31:0] dmemreq_bits_addr,   // Data mem port: read/write address
  output [31:0] dmemreq_bits_data,   // Data mem port: write data
  output        dmemreq_val,         // Data mem port: is dmem request valid?
  input  [31:0] dmemresp_bits_data   // Data mem port: returned read data

);
```

Figure 2: Interface for SMIPSv2 Processor

Test Harness

We are providing a test harness to connect to your processor model. The test harness is identical to the one described in *Tutorial 1: Simulating a SMIPSv1 Unpipelined Processor Using Synopsys VCS*. The test harness loads a SMIPS executable (VMH format) into the memory. The provided makefile can build both assembly tests as well as C benchmarks to run on your processor. The test harness will clock the simulation until it sees a non-zero value coming back on the `testrig_tohost` register, signifying that your processor has completed a test program. The `testrig_tohost` port should be set to zero on reset. A very simple test program is shown in Figure 3.

```
# 0x1000: Reset vector.
        addiu $2, $0, 1     # Load constant 1 into register r2
        mtc0  $2, $21       # Write tohost register in COP0
loop :  beq   $0, $0, loop  # Loop forever
```

Figure 3: Simple test program

Implemented Instructions

The SMIPS instruction set is a simplified version of the full MIPS32 instruction set. Consult the *SMIPS Processor Specification* for more details about the SMIPS architecture. You may also want to read *Tutorial 3: Assembly Programming for the SMIPS Processor*. For this first lab assignment, you will only be implementing the SMIPSv2 subset. Figure 5 shows the 35 instructions which make up the SMIPSv2 subset.

You do not need to support any exceptions or interrupt handling (apart from reset). The only pieces of the system coprocessor 0 you have to implement are the `tohost` and `fromhost` registers, and the MTC0 and MFC0 instructions that access these registers. These registers are used to communicate with the test harness. The test harness drives `testrig_fromhost`, while you should implement an 8-bit register in COP0 which drives the `testrig_tohost` port of the `smipsProc` module interface.

Getting Started

All of the 6.375 laboratory assignments should be completed on an Athena/Linux workstation. Please see the course website for more information on the computing resources available for 6.375 students. Once you have logged into an Athena/Linux workstation you will need to setup the 6.375 toolflow with the following commands.

```
% add 6.375
% source /mit/6.375/setup.csh
```

Note that to use the toolflow you will need to be using `tcsh`, the default shell on Athena/Linux. If you have switched to using `bash` you will need to change to `tcsh` before starting to use the 6.375 toolflow. Please ask the TA if you have any questions.

You will be using CVS to manage your 6.375 laboratory assignments. Please see *Tutorial 2: Using CVS to Manage Source Code* for more information on how to use CVS. Every student has their

own directory in the repository which is not accessible to other students. Assuming your Athena username is `cbatten`, you can checkout your personal CVS directory using the following command.

```
% cvs checkout 2006s/students/cbatten
```

To begin the lab you will need to make use of the lab harness located in `/mit/6.375/lab-harnesses`. The lab harness provides makefiles, scripts, and the Verilog test harness required to complete the lab. The following commands extract the lab harness into your CVS directory and adds the new project to CVS.

```
% cd 2006s/students/cbatten
% tar -xzvf /mit/6.375/lab-harnesses/lab1-harness.tgz
% find lab1 | xargs cvs add
% cvs commit -m "Initial checkin"
```

The resulting `lab1` project directory contains the following primary subdirectories: `src` contains your source Verilog; `tests` contains local assembly tests; `bmarks` contains local benchmarks; and `build` contains makefiles and scripts for building your processor. The `src` directory contains the `smipsTestHarness` Verilog module and various SMIPS instruction constants you may find helpful in `smipsInst.v`.

You can now begin to add your code to the `src` directory. You can use the following commands to build your simulator, run assembly level tests, run benchmarks in verification mode, and run benchmarks in performance mode. You will need to modify the makefile so that it has a listing of all your Verilog source files. Some of the tracing code in `smipsTestHarness` is specific to the staff's reference implementation, so you should feel free to modify it as needed. Consult *Tutorial 1: Simulating a SMIPSv1 Unpipelined Processor Using Synopsys VCS* for more information.

```
% cd lab1/build/vcs-sim-rtl
% make simv
% make run-asm-tests
% make run-bmarks-test
% make run-bmarks-perf
```

Your final lab submission should pass all of the assembly tests and also be able to successfully run the globally installed benchmarks. When you first start working on your processor it will not pass the multiply benchmark. This is because the multiply benchmark is incomplete. You will finish writing the benchmark in Question 4 of this lab assignment (see Section).

Lab Hints and Tips

This section contains several hints and tips which should help you in completing the lab assignment.

Tip 1: Use Incremental Development

We suggest taking an incremental approach when implementing the two-stage SMIPSv2 processor. Start by implementing just a few instructions and verify that they are working before slowly adding more instructions. Feel free to make use of the `examples/smipsv1-1stage-v` code

discussed in *Tutorial 1*. You can begin by moving some of the `smipsv1-1stage-v` code into your lab 1 project directory and verifying that it still passes the SMIPSv1 tests. Then create a two-stage SMIPSv1 processor and verifying that it can pass the five SMIPSv1 tests. The simplest approach for handling branches is for the fetch stage to always "predict" not-taken. Branches and jumps resolve in the execute stage; thus if the branch or jump was actually taken then you simply set the PC multiplexer appropriately and kill the instruction currently in the fetch stage. You can kill an instruction by inserting a NOP into the instruction register. Once you have a two-stage SMIPSv1 processor working, begin to refine the system into the two-stage SMIPSv2 system show in Figure 4 (for example you might want to create a separate branch address generator). After adding support for `lui` and `ori`, your processor should be able to pass the following SMIPSv2 tests: `smipsv2_simple.S`, `smipsv2_addiu.S`, `smipsv2_bne.S`, `smipsv2_lw.S`, `smipsv2_sw.S`, `smipsv2_ori.S`, and `smipsv2_lui.S`.

Now you can gradually add additional instructions and attempt to incrementally pass more of the assembly test suite. We strongly discourage implementing the entire system and all instructions before trying to pass any tests. A more incremental approach will greatly reduce your verification time.

Tip 2: Make Use of the Verilog Component Library

We have provided you with a very simple Verilog Component Library (VCLIB) which you may find useful for this lab. VCLIB includes simple mux, register, arithmetic, and memory modules. It is installed globally in the locker at `/mit/6.375/install/vclib`. Examine the makefile included in the lab harness to see how to link in the library.

Tip 3: Use a Semi-Behavioral ALU

Consider using a more behavioral implementation of the ALU. Instead of trying to design an optimized gate-level ALU, use a conditional statement to select the result from among the various built in Verilog operators. For example, an incomplete ALU module might use the following Verilog.

```
assign out = ( fn == 4'd0 ) ? ( in0 +  in1 )
           : ( fn == 4'd1 ) ? ( in0 -  in1 )
           : ( fn == 4'd3 ) ? ( in0 < in1 )
           : ( fn == 4'd4 ) ? ( $signed(in0) < $signed(in1) )
           : ( fn == 4'd5 ) ? ( $signed(in1) >>> in0[4:0] )
           : 32'bx;
```

It would not be surprising if this code synthesized into five separate arithmetic blocks (an adder, a subtracter, two comparators, and a shifter) and an output multiplexer. This is obviously not the desired hardware; we would like to use the same adder for the addition, subtraction, and comparison operations. Although not an ideal representation of the desired hardware, this ALU is still synthesizable making it "semi-behavioral". It makes an excellent initial ALU implementation. Implementing signed arithmetic in Verilog can be tricky. The above example shows one way to implement a signed comparison and an arithmetic right shift.

Notice that in the system diagram shown in Figure 4, LUI would be implemented by using the ALU to shift the zero-extended immediate 16 bits to the left. You can select the constant 16 in the operand 1 mux to act as the shift amount for this shift.

Tip 4: Handle Reset Carefully

It is very important to plan how you will handle reset. Figure 4 highlights those state elements which you should probably reset. The *SMIPS Processor Specification* states that the `tohost` register must be reset to zero. Ask yourself what should the PC and the IR be reset to? If you would like, you can use the `vcRDFF_pf` module in the VCLIB for these state elements.

Tip 5: Use Text Tracing Wisely

When you first try compiling with the lab 1 harness you will probably see several "Cross module resolution failed" VCS errors. These errors are because the `smipsTestHarness` module includes references to signals and nets that are in the staff's implementation of lab 1 but are probably not in your implementation. They are there to illustrate a technique for text tracing. We recommend adding some text tracing to your `smipsTestHarness` such that value of various nets in your processor are displayed each cycle - one cycle per line. See the `smipsv1-1stage-v` code for an example of text tracing. A common debugging technique is to first try running a test program on your processor. From examining the text trace you should be able to get a good feel for how your processor is executing. If you need to do more detailed debugging, then start VirSim and use the waveform viewer to trace more signals. See *Tutorial 1* for more on using VirSim.

Tip 6: Run Tests Individually

Although the makefile contains convenient targets for running all of the assembly tests at once, when you are initially debugging your processor you will want to get a simple test working before trying all of the tests. The following commands will build the simulator, build the `smipsv1_simple.S` test, and then run the test on the simulator. After running the test you can open the `test.out` file to examine the text trace output. Each assembly test is a self-checking test. If the test passes, it will write one to the `tohost` register and the test harness will stop and print `*** PASSED ***`. If the test fails, it will write a value greater than one to the `tohost` register and the test harness will stop and print `*** FAILED ***`. For SMIPSv2 tests, the value in the `tohost` register corresponds to which test case in the assembly test failed. Consult the appropriate SMIPSv2 assembly test file to learn more about the specific test case.

```
% make simv
% make smipsv1_simple.S.vmh
% ./simv +exe=smipsv1_simple.S.vmh >! test.out
```

Tip 7: Use the SMIPS Assembly and Objdump Files for Debugging

When debugging your processor, you should consult both the SMIPS assembly file as well as the objdump file. The assembly test files are installed globally in the locker at `/mit/6.375/install/smips-tests`. When you build the `smipsv1_simple.S` test it will also generate a `smipsv1_simple.S.dump` file. The objdump file shows the exact instructions (and their addresses) which make up the corresponding assembly test. By examining which PC's your processor is executing and correlating this to the objdump and assembly file you should be able to figure out what your processor is doing.

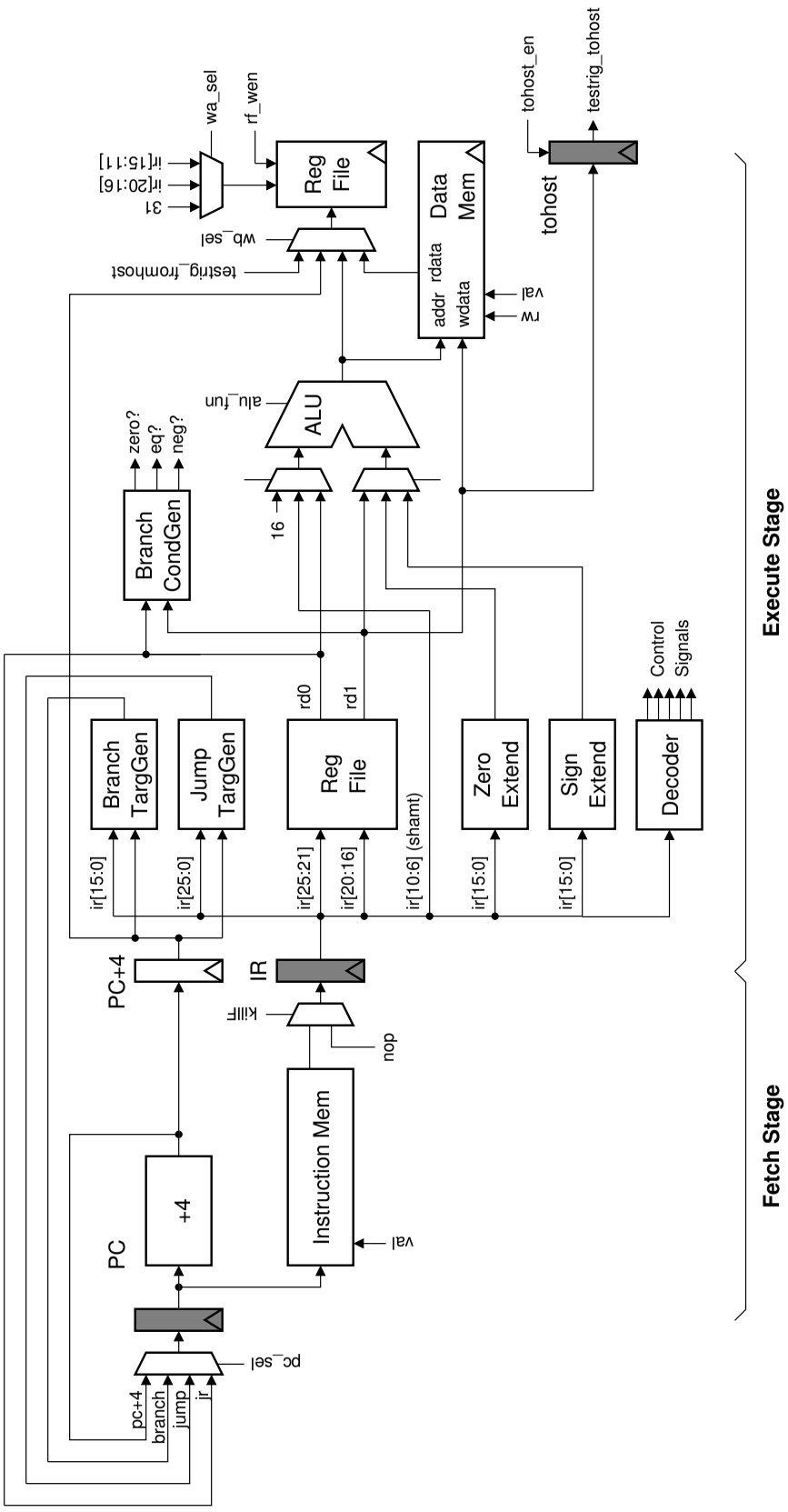

Figure 4: Two-Stage Pipeline for SMIPSv2 Processor. Shaded state elements need to be correctly loaded on reset.

31 26	25 21	20 16	15 11	10 6	5 0	
opcode	rs	rt	rd	shamt	funct	R-type
opcode	rs	rt	immediate			I-type
opcode	target					J-type
Load and Store Instructions						
100011	base	dest	signed offset			LW rt, offset(rs)
101011	base	dest	signed offset			SW rt, offset(rs)
I-Type Computational Instructions						
001001	src	dest	signed immediate			ADDIU rt, rs, signed-imm.
001010	src	dest	signed immediate			SLTI rt, rs, signed-imm.
001011	src	dest	signed immediate			SLTIU rt, rs, signed-imm.
001100	src	dest	zero-ext. immediate			ANDI rt, rs, zero-ext-imm.
001101	src	dest	zero-ext. immediate			ORI rt, rs, zero-ext-imm.
001110	src	dest	zero-ext. immediate			XORI rt, rs, zero-ext-imm.
001111	00000	dest	zero-ext. immediate			LUI rt, zero-ext-imm.
R-Type Computational Instructions						
000000	00000	src	dest	shamt	000000	SLL rd, rt, shamt
000000	00000	src	dest	shamt	000010	SRL rd, rt, shamt
000000	00000	src	dest	shamt	000011	SRA rd, rt, shamt
000000	rshamt	src	dest	00000	000100	SLLV rd, rt, rs
000000	rshamt	src	dest	00000	000110	SRLV rd, rt, rs
000000	rshamt	src	dest	00000	000111	SRAV rd, rt, rs
000000	src1	src2	dest	00000	100001	ADDU rd, rs, rt
000000	src1	src2	dest	00000	100011	SUBU rd, rs, rt
000000	src1	src2	dest	00000	100100	AND rd, rs, rt
000000	src1	src2	dest	00000	100101	OR rd, rs, rt
000000	src1	src2	dest	00000	100110	XOR rd, rs, rt
000000	src1	src2	dest	00000	100111	NOR rd, rs, rt
000000	src1	src2	dest	00000	101010	SLT rd, rs, rt
000000	src1	src2	dest	00000	101011	SLTU rd, rs, rt
Jump and Branch Instructions						
000010	target					J target
000011	target					JAL target
000000	src	00000	00000	00000	001000	JR rs
000000	src	00000	dest	00000	001001	JALR rd, rs
000100	src1	src2	signed offset			BEQ rs, rt, offset
000101	src1	src2	signed offset			BNE rs, rt, offset
000110	src	00000	signed offset			BLEZ rs, offset
000111	src	00000	signed offset			BGTZ rs, offset
000001	src	00000	signed offset			BLTZ rs, offset
000001	src	00001	signed offset			BGEZ rs, offset
System Coprocessor (COP0) Instructions						
010000	00000	dest	cop0src	00000	000000	MFC0 rt, rd
010000	00100	src	cop0dest	00000	000000	MTC0 rt, rd

Figure 5: SMIPSv2 Instruction Set

Critical Thinking Questions

The primary deliverable for this lab assignment is your working Verilog RTL for the two-stage SMIPSv2 processor. In addition, you should prepare written answers to the following questions and turn them in at the beginning of class on February 24. Although you should attempt Question 3, it could require quite a bit of design work. Do not worry if you are unable to finish Question 3. As long as you make a reasonable effort you will not be penalized. It is challenging, but it servers to illustrate the complexities which can arise when designing hardware which is more complicated than the simple two-stage pipe.

Question 1: Design Partitioning

We discussed in lecture the importance of partitioning your design for semi-custom toolflows. Clearly identify on the system diagram shown in Figure 4 which components you placed in your datapath and which components you placed in your control logic. Also clearly highlight all of your control signals including any additional signals which you may have added. If your system differs significantly from the one shown in Figure 4 then draw your own system diagram. Please provide a very brief reasoning for your why you decided to place the instruction register and register write address mux in either the datapath or the control unit.

Question 2: Physical Design Predictions

In your opinion, what will be the most critical paths in the design? Select what you think will be the two most critical paths and either draw them on the datapath diagram or provide a textual description of the paths. What component do you think will take up the most area in the design? You will be able to test your hypotheses in Lab 2.

Question 3: Adding a Simple Branch Predictor

This question requires you to make some modifications to your design and to evaluate the impact those changes have on the performance of your processor. The current two-stage SMIPSv2 design incurs a one-cycle penalty for every taken branch. Your first task is to evaluate the impact this has on the overall performance of your processor. For this question, we will be measuring performance in terms of the number of instructions which are executed per cycle (IPC). Obviously, the IPC of your design cannot exceed one. It can, however, be significantly less than one due to the single cycle taken branch delay penalty. Use the following commands to measure the IPC of your processor.

```
% cd lab1/build/vcs-sim-rtl
% make run-bmarks-perf
```

This should run the installed global benchmarks on your processor and display the IPC for each. You can find IPC as well as other statistics in the corresponding `benchmark.out` file. Report these IPC numbers in your answer to this question.

Your second task is to add a simple branch predictor to your original SMIPS processor. For more information on branch predictors please consult *Computer Organization and Design: The Hardware/Software Interface*, by Patterson and Hennessey or the 6.823 course lecture slides. The TAs would also be happy to chat with you about branch predictors.

Make your changes in such a way that you can build your processor both with and without the branch predictor. For example, you might have two versions of your datapath, control logic, and top-level `smipsProcessor` module - one with and one without the predictor. You can then use two different build directories: use `build` to build your processor without the branch predictor and use `build-bpred` to build your processor with the branch predictor.

We are now going to sketch a simple predictor. We recommend that you start by implementing this simple predictor. If you have time and are interested, you are welcome to improve the predictor.

You will be implementing a simple direct-mapped branch-target-buffer (BTB). We will *not* be predicting JR or JALR instructions. These instructions should be handled exactly as they are in the baseline design: squash the fetch stage when taken. Figure 6 shows the modified system diagram. The predictor should contain a small (four entry) table. Each entry in the table is a <pc+4,target> pair. You will also need some form of valid bits so that you can correctly handle uninitialized table entries. You might want to make use of the `vcRAM_rst_1w1r_pf` module in VCLIB for your valid bits. The operation of the predictor is described below.

In the fetch stage, the predictor uses the low order bits of `pc+4` to index into the table. We use `pc+4` instead of `pc` because it makes the pipelining simpler. The predictor should read out the corresponding <pc+4,target> pair from the table, and if the pc's match then this is a hit. We then choose the proper `pc_sel` signal so that the predicted target gets clocked into the PC on the next cycle. If the pc's do not match, then this is a miss and we use `pc+4` as the next pc. Our simple predictor uses the following invariant: if a pc is in the table then it is always predicted taken, but if a pc is not in the table then we always predict not-taken. Entries are never removed from the table they are only overwritten. Since we are not predicting JR and JALR, we know that the target address is *always* correct even if our taken/non-taken prediction is incorrect. We do not need to verify the target address, only the taken/not-taken prediction.

We pipeline the predictor hit/miss signal to the execute stage. Because of the invariant mentioned above, this hit/miss bit also tells us if the branch was predicted taken or not-taken. In the execute stage, the predictor should compare the predicted taken/non-taken bit to the calculated taken/not-taken bit. This is how the predictor can determine if there as a misprediction. There are four possible scenarios shown in the following table.

Predicted	**Actual**	**Mispredict?**	**Action to take**
taken	taken	no	no action required
not-taken	taken	yes	kill instr in fetch, update table, pc := branch or jump targ
taken	non-taken	yes	kill instr in fetch, do not update table, pc := correct pc+4
not-taken	not-taken	no	no action required

If the branch was predicted not-taken, and it was actually taken (i.e. we enter a loop), then we update the table by adding the appropriate `pc+4` and branch or jump target. This corresponds to writing the `correct targ` and `correct pc+4` signals shown in Figure 6. If the branch was predicted taken and it was actually not-taken (i.e. we fall out of a loop), then we do *not* update the table. We could invalidate the appropriate entry in the table, but to make things simpler we just leave the table unchanged.

There are several subtle issues to be aware of when implementing the predictor. The most important is to carefully think about the situation when there are back-to-back branches. For example, what

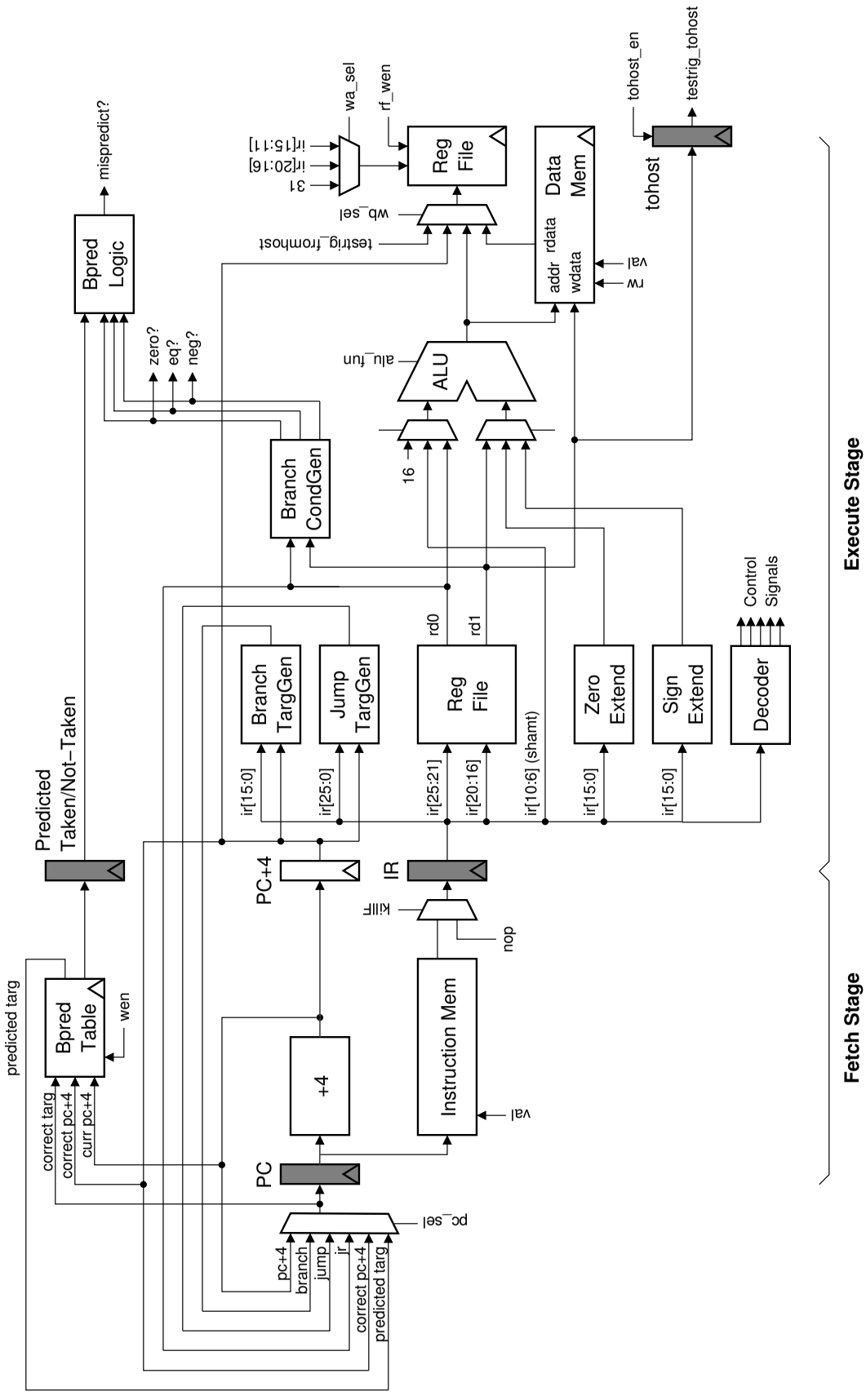

Figure 6: Two-Stage Pipeline for SMIPSv2 Processor with Branch Predictor.

happens if the execute stage identifies a misprediction, but at the same time there is a branch in the fetch stage which is being predicted taken? It is exactly these type of difficult concurrency issues which Bluespec helps manage.

If you are feeling particularly ambitious there are several ways to improve on this simple design including: adding support for JR and JALR prediction, using a set-associative table instead of a direct mapped table, increasing the size of the table, or adding some hysteresis to the table (i.e. it takes more than one taken branch before we predict taken).

For this question, you should submit a description of your branch predictor and the IPC results for your design both with and without the predictor.

Question 4: Analyzing a Simple SMIPS Benchmark

For this question you will first write a small SMIPS assembly routine, and then evaluate ISA changes which would affect the performance of that routine. The SMIPSv2 ISA does not include a multiply instruction, yet we can emulate multiplication using shifts and adds. The following pseudo code illustrates a straightforward algorithm for software multiplication.

```
function multiply ( op1, op2 ) {
  r1 := op1
  r2 := op2
  r3 := 0

  for ( i = 0; i < 32; i++ ) {
    if ( ( r1 & 0x1 ) == 1 )
      r3 := r3 + r2
    r1 := r1 >> 1
    r2 := r2 << 1
  }

  return r3
}
```

Your first task is to write a software multiplication routine in SMIPS assembly. We have provided you with a C test program and SMIPS assembly template. You can find a C version of the multiply routine in `bmarks/multiply/multiply.c`. Modify `bmarks/multiply/multiply_asm.S` by adding your SMIPS code where indicated. You can then build and test your multiply benchmark using the following commands.

```
% cd lab1/build/vcs-sim-rtl
% make simv
% make multiply.test.vmh
% ./simv +exe=multiply.test.vmh +max-cycles=40000
```

Once your multiply routine is working you can evaluate the performance (measured in IPC) with the following commands.

```
% make multiply.perf.vmh
% ./simv +exe=multiply.test.vmh +max-cycles=40000 +verify=0
```

The resulting output will show some statistics including the IPC. In your answer to this question, report the IPC of this benchmark with and with out your branch predictor from Question 4. The multiply routine has two branches: the outer for loop branch and the inner if statement branch. For which branch will your predictor generate good predictions? For which branch will the predictor generate poor predictions?

An SMIPS assembly programmer approaches you and suggests that you add the following conditional add instruction to the SMIPS ISA. The syntax and semantics for this new instruction are shown below.

`addu.c r3, r2, r1` if ((r1 & 0x1) == 1) r3 := r3 + r2

Would this new instruction help? Why? What changes would you need to make to your SMIPSv2 datapath in order to support this new instruction? Do you think these changes would impact the cycle time or area of your design?

Optional Bonus Question

The MIPS32 ISA uses `pc+4` for the branch and jump target address. Using `pc+4` instead of `pc` seems arbitrary but it enables a clever designer to eliminate the `pc+4` pipeline register. Can you figure out how to remove the `pc+4` register while still enabling correct execution? Be careful about back-to-back branches!